Formalisation en Coq d'algorithmes de filtres numériques

Diane Gallois-Wong

Formalisation en Coq d'algorithmes de filtres numériques

Diane Gallois-Wong[1,2⋆]

[1] Université Paris-Sud
[2] LRI, CNRS & Univ. Paris-Sud, Université Paris-Saclay, bâtiment 650, Université Paris-Sud, F-91405 Orsay Cedex, France

Abstract. Les algorithmes de filtres numériques sont un outil essentiel du traitement du signal et du contrôle-commande. Ils sont utilisés dans de nombreux domaines comme les télécommunications, l'automobile, la robotique, l'aéronautique etc. Comme certains de ces domaines sont critiques, nous souhaitons garantir formellement qu'ils se comportent bien numériquement malgré les erreurs d'arrondi inhérentes à leur implémentation en précision finie. Or il existe de nombreux algorithmes pour calculer un filtre numérique, que l'on appelle réalisations. En précision finie, des réalisations légèrement différentes, par exemple où seulement l'ordre des calculs est modifié, peuvent donner des résultats différents. La SIF (*Specialized Implicit Form*) est un formalisme très général, qui peut représenter toutes les réalisations en détaillant jusqu'à l'ordre des calculs. Cet article étend la formalisation en Coq des filtres numériques présentée dans [1] : il y ajoute la définition de la SIF, les traductions des réalisations de cette formalisation vers la SIF, et le théorème du filtre d'erreur associé à une SIF, qui décrit la propagation des erreurs de calcul au fil des itérations dans l'algorithme correspondant.

1 Introduction

Les filtres numériques sont utilisés dans de nombreux contextes, du traitement du signal au contrôle-commande, du lecteur MP3 au système de contrôle d'un avion. Ils permettent de transformer les signaux numériques, qui sont des suites de valeurs prises par des grandeurs physiques au cours du temps, en d'autres signaux numériques. Ils peuvent ainsi amplifier un signal sonore, ou encore produire la suite de commandes à fournir à l'avion à partir de toutes les données mesurées par ses capteurs. Implémenter un filtre numérique nécessite l'utilisation d'une arithmétique en précision finie, généralement virgule flottante ou virgule fixe, ce qui implique des erreurs d'arrondi. De plus, les algorithmes de filtres numériques sont fortement itératifs. D'une part, cela introduit un risque d'accumulation de nombreuses erreurs qui paraissent négligeables en une erreur critique. D'autre part, cela rend l'étude de ces erreurs d'arrondi complexe : chaque erreur peut influencer toutes les itérations futures.

⋆ Ce travail a bénéficié du soutien financier de la "Fondation CFM pour la Recherche".

Cet article s'inscrit dans la continuité de [1], une formalisation en Coq des filtres numériques qui contient les premières étapes vers une analyse formelle complète des erreurs d'arrondi. Cette formalisation définit les filtres linéaires invariants dans le temps, la famille usuelle de filtres que nous souhaitons étudier. Elle présente plusieurs algorithmes de filtres utilisés en pratique, appelés réalisations. Elle prouve deux théorèmes essentiels à cette analyse d'erreurs. Le théorème du filtre d'erreur décrit la propagation des erreurs au fil des calculs pour une réalisation appelée State-Space. Le théorème du Worst-Case Peak-Gain permet de borner le signal de sortie d'un filtre lorsqu'on connaît une borne sur l'entrée, et son application permet justement de borner l'erreur finale déterminée par le théorème précédent.

Le sujet principal de cet article est une formalisation[3] en Coq [2,3] de la SIF (*Specialized Implicit Form*) [4], un formalisme qui permet de décrire tous les algorithmes de filtres numériques linéaires invariants dans le temps, en explicitant l'ordre dans lequel les calculs sont effectués. Nous définissons en Coq une traduction d'un State-Space vers une SIF, en prouvant que le filtre implémenté est conservé. Cela nous permet de décrire sous forme de SIF plusieurs réalisations usuelles ayant été traduites vers un State-Space dans [1]. Enfin, nous prouvons formellement le théorème du filtre d'erreur dans le cadre d'un filtre défini par une SIF. Comme dans [1], nous ne considérons pour l'instant pas de standard particulier pour les nombres en précision finie. Nous travaillons dans le cadre de n'importe quelle arithmétique en précision finie, avec des nombres réels modifiés par des termes d'erreur arbitraires. Choisir une de ces arithmétiques, et obtenir ainsi plus d'informations sur ces termes d'erreur, sera une étape future naturelle.

Notre formalisation en Coq étant un surensemble de celle de [1], elle utilise les mêmes bibliothèques et axiomes. Nous considérons les réels axiomatiques de la bibliothèque standard [5]. Nos vecteurs et matrices sont basés sur ceux de la bibliothèque d'analyse réelle Coquelicot [6]. Deux axiomes nous permettent de manipuler facilement fonctions et preuves, et sont généralement acceptés comme étant sûrs par la communauté. D'une part, `FunctionalExtensionality` affirme que deux fonctions prenant les mêmes valeurs sur toutes les entrées sont égales. D'autre part, `ProofIrrelevance` énonce que deux preuves d'une même propriété sont égales.

Différentes études formelles des erreurs d'arrondi dans les filtres numériques ont déjà été menées. Akbarpour et Tahar prouvent en HOL un résultat similaire au théorème du filtre d'erreur dans le cadre de plusieurs réalisations canoniques [7]. Akbarpour *et al.* comparent des implémentations de filtres en virgule flottante et en virgule fixe, sous l'hypothèse que les opérations arithmétiques ne peuvent pas produire d'*overflow* [8]. Park *et al.* bornent les erreurs d'arrondi pour une itération de la boucle principale d'une algorithme de filtre, mais n'étudient pas leur propagation [9]. Grâce à la SIF, notre formalisation couvre tous les algorithmes de filtres numériques et tient compte de l'ordre des calculs. Le théorème de filtre d'erreur que nous prouvons décrit la propagation des erreurs d'arrondi pour n'importe quelle arithmétique en précision finie.

[3] Fichiers disponibles à `www.lri.fr/~gallois/code/coq-filtresnum-JFLA19.tgz`

La section 2 définit les filtres numériques et quelques réalisations usuelles. Son contenu a été intégralement formalisé en Coq dans [1]. Nous reprenons simplement ses éléments les plus pertinents pour les deux sections suivantes, dont les formalisations sont originales. La section 3 présente la SIF, son intérêt, sa définition en Coq et des correspondances prouvées entre celle-ci et le State-Space. La section 4 présente le théorème du filtre d'erreur dans le cas d'un filtre décrit par une SIF. Enfin, la section 5 conclut et propose quelques perspectives.

Notations. Nous écrirons les vecteurs en gras (et en minuscule), les matrices en gras et en majuscule : ainsi $\boldsymbol{a}$ représentera un vecteur et $\boldsymbol{A}$ une matrice, tandis que a sera un scalaire. Nous noterons $\boldsymbol{I}$ une matrice identité et $\mathbf{0}$ une matrice nulle de dimensions adaptées au contexte, ainsi que $\boldsymbol{I}_n$ la matrice identité de taille explicite $n \times n$.

2 Filtre numérique

Cette section présente les éléments de traitement du signal nécessaires pour comprendre le fonctionnement et l'intérêt de la SIF présentée en section 3. Elle introduit aussi les définitions Coq correspondantes les plus essentielles. Celles-ci sont directement reprises de [1], qui fournit des explications plus détaillées, notamment concernant les choix techniques.

2.1 Signal numérique

Un signal numérique représente l'évolution discrète d'une grandeur physique (vitesse, température, position etc.) au cours du temps. Un signal numérique est donc défini mathématiquement comme une suite de nombres réels, ou de vecteurs réels (dans le cas d'une position par exemple). On parle de signal réel ou signal vectoriel pour distinguer ces deux possibilités. Cette suite est indexée sur un intervalle de nombres entiers, fréquemment $\mathbb{Z}$, $\mathbb{N}$ ou de la forme $\{0, 1, ..., N\}$. Nous avons choisi de considérer n'importe quel indice dans $\mathbb{Z}$, mais avec la restriction que le signal doit prendre la valeur zéro (ou vecteur nul) pour les indices strictement négatifs : pour tout signal x et entier $k < 0$, $x(k) = 0$. C'est équivalent à définir les signaux seulement sur $\mathbb{N}$, mais présente des intérêts liés à l'initialisation et à la simplicité visuelle des énoncés de théorèmes, discutés dans [1].

Les définitions Coq sont reprises de [1]. Un signal réel se compose d'une fonction de $\mathbb{Z}$ dans $\mathbb{R}$ accompagnée de la preuve qu'elle vaut zéro pour $k < 0$. Un signal vectoriel est défini similairement. Les vecteurs `vect n` sont basés sur ceux de la bibliothèque Coquelicot mais leurs indices sont des entier relatifs (une valeur par défaut étant renvoyée pour un indice non compris entre 1 et la taille `n`).

```
Definition causal (x : Z → R) := (forall k : Z, (k < 0)%Z → x k = 0%R).
Record signal := { signal_val :> Z → R ; signal_prop : causal signal_val }.
Context { n : Z }.
Definition vect_causal (x : Z → vect n) := forall k, (k < 0)%Z → x k = vect_0.
Record vect_signal := { vect_signal_val :> Z → vect n ;
                        vect_signal_prop : vect_causal vect_signal_val }.
```

On définit les trois opérations élémentaires suivantes sur les signaux (présentées avec des signaux réels mais facilement étendues à des vecteurs, en notant que c reste un réel). Les définir en Coq nécessite de prouver que le résultat est encore un signal (nul pour les indices strictement négatifs).

- L'addition (terme à terme) : $y = x_1 + x_2$ signifie $\forall k,\ y(k) = x_1(k) + x_2(k)$.
- La multiplication par une constante $c \in \mathbb{R}$: $y = c\,x$ signifie $\forall k,\ y(k) = c\,x(k)$.
- Le retard, ou translation dans le temps : "y est le signal x retardé de $K \geq 0$ unités de temps" signifie $\forall k,\ y(k) = x(k - K)$.
  Noter que l'initialisation imposée sur nos signaux implique $\forall k < K,\ y(k) = 0$.
  En Coq, on choisit de renvoyer arbitrairement le signal identiquement nul lorsque $K < 0$, car il est plus facile de travailler avec des fonctions totales.

2.2 Filtres LTI

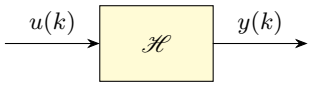


Fig. 1: Filtre numérique.

Un filtre, généralement noté $\mathscr{H}$, est une fonction qui transforme un signal d'entrée u en un signal de sortie $y = \mathscr{H}\{u\}$. Ceci est représenté par la figure 1. La sortie $y(k)$ à l'instant k peut dépendre de l'entrée correspondante $x(k)$, mais aussi de toutes les entrées antérieures (mais évidemment pas des entrées futures dans les filtres qui sont étudiés en pratique). Un filtre est appelé SISO (*Single-Input Single-Output*) s'il transforme des signaux réels en signaux réels, ou MIMO (*Multiple-Input Multiple-Output*) pour des signaux vectoriels.

```
Definition filter := signal -> signal.
Definition MIMO_filter {N_in N_out : Z} :=
  @vect_signal N_in -> @vect_signal N_out.
```

Nous nous intéressons ici à une famille de filtres très utilisée en pratique : les filtres linéaires invariants dans le temps (LTI pour *Linear Time Invariant*), qui satisfont les propriétés suivantes (avec des quantifications universelles sur les signaux qui peuvent être réels ou vectoriels, et sur la constante $c \in \mathbb{R}$) :

- Compatibilité avec l'addition : $\mathscr{H}\{u_1 + u_2\} = \mathscr{H}\{u_1\} + \mathscr{H}\{u_2\}$
- Compatibilité avec la multiplication par une constante : $\mathscr{H}\{c\,u\} = c\,\mathscr{H}\{u\}$
- Invariance dans le temps, i.e. si l'entrée est retardée de $K \geq 0$ unités de temps alors la sortie l'est aussi :
  $$\forall k,\ u'(k) = u(k - K) \quad \Longrightarrow \quad \forall k,\ \mathscr{H}\{u'\}(k) = \mathscr{H}\{u\}(k - K)$$

2.3 Différentes réalisations

Il existe beaucoup de familles d'algorithmes de filtres, appelées réalisations. Un même filtre peut être exprimé par diverses réalisations, qui calculent toutes la même relation entrée-sortie lorsqu'on considère des nombres réels. Cependant, en précision finie, le choix de la réalisation est très important : en plus des différences de complexité en temps et en mémoire, apparaissent des différences de comportement numérique. Ainsi, pour un filtre donné, selon la réalisation et l'arithmétique en précision finie utilisées, la sortie calculée en pratique pourra plus ou moins s'écarter de la sortie du modèle en précision infinie.

La réalisation la plus classique est l'équation aux différences, aussi appelée forme directe I (DFI pour *Direct Form I*) :

$$\forall k \geq 0,\ y(k) = \sum_{i=0}^{n} b_i u(k-i) - \sum_{i=1}^{n} a_i y(k-i) \tag{1}$$

Les $(a_i)_{1 \leq i \leq n}$ et $(b_i)_{0 \leq i \leq n}$ sont les coefficients du filtre, et n est appelé l'ordre du filtre. Ce sont ces mêmes coefficients qui apparaissent dans la fonction de transfert, un objet mathématique permettant de décrire les filtres dans le domaine fréquentiel. Mais cet article s'intéresse seulement au domaine temporel et aux algorithmes itératifs associés. Par exemple, la forme directe I permet naturellement de calculer chaque sortie $y(k)$ à condition d'avoir gardé en mémoire les entrées et sorties des n étapes précédentes.

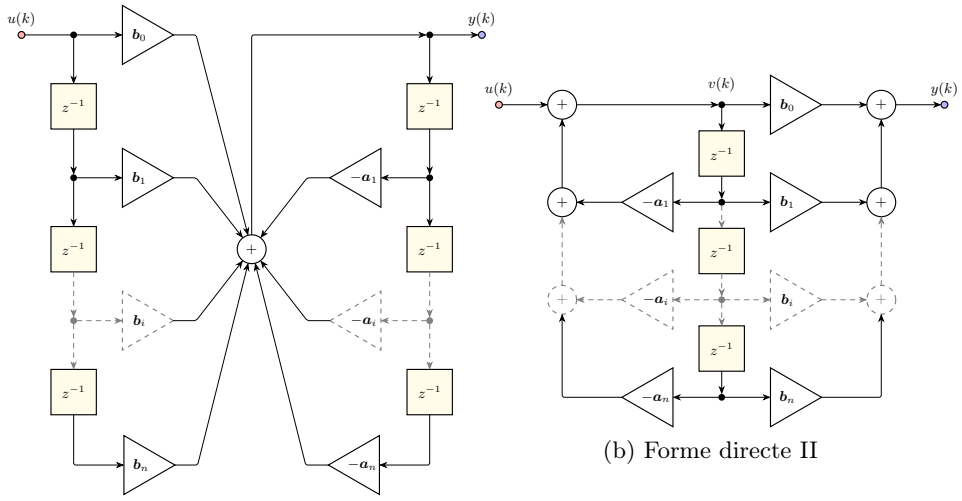
La forme directe II (DFII pour *Direct Form II*) utilise aussi les coefficients de la fonction de transfert :

$$\forall k \geq 0, \quad \begin{cases} e(k) = u(k) - \sum_{i=1}^{n} a_i e(k-i) \\ y(k) = \sum_{i=1}^{n} b_i e(k-i) \end{cases} \tag{2}$$

où e est un signal auxiliaire. À chaque étape, on peut calculer les valeurs de e et y en ayant retenu seulement les n valeurs précédentes de e, ce qui fait deux fois moins de mémoire nécessaire que pour la forme directe I.

Les formes directes I et II peuvent être représentées par les graphes à flot de données de la figure 2. Les noeuds marqués d'un + sont bien sûr des additions, et les triangles des multiplications par une constante. Un z^{-1} représente un retard d'une unité de temps : à chaque étape, il produit la valeur qu'il a reçue à l'étape précédente. Le nombre de valeurs à garder en mémoire correspond au nombre de retards dans le graphe : $2n$ pour la forme directe I, n pour la forme directe II.

Le *State-Space* est une réalisation plus générale, couramment utilisée par la communauté de l'automatique. Contrairement aux formes directes précédentes, il permet de représenter un filtre MIMO. Le State-Space utilise un signal auxiliaire $\boldsymbol{x}$, appelé vecteur d'état, pour stocker toutes les informations dont on a besoin pour calculer la sortie courante, mais aussi les sorties futures. La relation entre



(a) Forme directe I

(b) Forme directe II

Fig. 2: Graphes à flot de données des formes directes I et II.

l'entrée $\boldsymbol{u}$ et la sortie $\boldsymbol{y}$ est la suivante (les signaux étant bien sûr nuls pour $k < 0$) :

$$\begin{aligned} \boldsymbol{x}(k+1) &= \boldsymbol{A}\boldsymbol{x}(k) + \boldsymbol{B}\boldsymbol{u}(k) \\ \boldsymbol{y}(k) &= \boldsymbol{C}\boldsymbol{x}(k) + \boldsymbol{D}\boldsymbol{u}(k) \end{aligned} \tag{3}$$

où les matrices $\boldsymbol{A}$, $\boldsymbol{B}$, $\boldsymbol{C}$ et $\boldsymbol{D}$ sont des coefficients qui caractérisent le filtre.

N'importe quelle réalisation décrivant un filtre MIMO peut représenter un filtre SISO en considérant des vecteurs d'entrée $\boldsymbol{u}(k)$ et de sortie $\boldsymbol{y}(k)$ de taille 1. Dans le cas du State-Space, $\boldsymbol{B}$ devient alors un vecteur colonne, $\boldsymbol{C}$ un vecteur ligne, et $\boldsymbol{D}$ une matrice de taille 1×1, qu'on peut identifier à un scalaire. En revanche, le vecteur d'état $\boldsymbol{x}(k)$ peut quand même être de taille strictement supérieure à 1 : même pour un filtre SISO, on peut avoir besoin de stocker de multiples valeur d'une étape sur l'autre.

Le State-Space est très général : on peut transformer n'importe quelle réalisation en un State-Space qui représente le même filtre, c'est-à-dire la même relation entrée-sortie. Par exemple, la forme directe I est facilement transformée en un State-Space en sauvegardant les n sorties précédentes dans le vecteur d'état. Cependant, la traduction d'un algorithme vers un State-Space risque de modifier l'ordre des calculs, comme dans l'exemple de la section 3.2. Dans ce cas, même si les filtres théoriques représentés sont les mêmes, les sorties calculées en pratique peuvent être différentes. La SIF, présentée en section 3, est une extension du State-Space qui résout ce problème car elle est capable de préserver l'ordre des calculs de n'importe quel algorithme.

Ces trois réalisations (forme directe I, forme directe II et State-Space) et leur formalisation en Coq, qui inclut des traductions des deux premières vers le State-Space, sont discutées avec plus de détails dans [1].

3 La SIF : utilité et formalisation

Il existe de nombreuses réalisations de filtres numériques en plus des formes directes I et II et du State-Space présentés en section 2.3 : formes directes transposées, opérateur ρ, structures LCW et LGS, décomposition en cascade ou en parallèle, etc. [10] Chacune présente ses propres avantages en termes de complexité en temps et en mémoire, compatibilité logicielle et matérielle, et comportement numérique pour une arithmétique en précision finie donnée. Cependant, formaliser l'analyse d'erreurs d'arrondi séparément pour chacune des nombreuses réalisations utilisées en pratique représenterait un travail très long. À la place, nous formalisons la SIF (*Specialized Implicit Form*) [4], une réalisation canonique qui permet de décrire n'importe quelle autre réalisation, en conservant l'ordre dans lequel les calculs sont effectués. Ce dernier point est très important, car l'ordre des opérations modifie les erreurs d'arrondi que nous souhaitons étudier. Ainsi, les propriétés prouvées pour la SIF, notamment le théorème du filtre d'erreur de la section 4, peuvent être appliquées à n'importe quelle autre réalisation, à condition de traduire formellement celle-ci en SIF.

En pratique, une SIF n'est pas implémentée directement sous forme de calcul matriciel. Cependant, on déduit facilement d'une SIF une implémentation sous forme de séquence d'affectations scalaires comportant principalement des sommes de produits.

3.1 Définition de la SIF et ordre des calculs

La SIF est une extension du State-Space. Une SIF est caractérisée par neuf matrices $(\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$ au lieu de quatre pour le State-Space. De plus, la matrice carrée $\boldsymbol{J}$ doit toujours être triangulaire inférieure, avec des 1 sur toute la diagonale principale : par exemple, si elle est de taille 3×3, elle aura la forme $\begin{pmatrix} 1 & 0 & 0 \\ ? & 1 & 0 \\ ? & ? & 1 \end{pmatrix}$. Cette condition sera appelée *Jprop*, et implique notamment que $\boldsymbol{J}$ est inversible. Le filtre décrit par une SIF est donné par la relation entrée $\boldsymbol{u}$ - sortie $\boldsymbol{y}$ suivante :

$$\begin{aligned} \boldsymbol{J}\boldsymbol{t}(k+1) &= \boldsymbol{M}\boldsymbol{x}(k) + \boldsymbol{N}\boldsymbol{u}(k) \\ \boldsymbol{x}(k+1) &= \boldsymbol{K}\boldsymbol{t}(k+1) + \boldsymbol{P}\boldsymbol{x}(k) + \boldsymbol{Q}\boldsymbol{u}(k) \\ \boldsymbol{y}(k) &= \boldsymbol{L}\boldsymbol{t}(k+1) + \boldsymbol{R}\boldsymbol{x}(k) + \boldsymbol{S}\boldsymbol{u}(k) \end{aligned} \tag{4}$$

Le signal auxiliaire $\boldsymbol{x}$ est toujours le vecteur d'état, mais il y a un deuxième signal auxiliaire $\boldsymbol{t}$, qui sert à calculer des valeurs intermédiaires et qu'on appellera vecteur intermédiaire. C'est la présence de $\boldsymbol{t}$ qui distingue la SIF du State-Space.

En effet, si on enlevait toutes les apparitions de $\boldsymbol{t}$, c'est-à-dire la première ligne entière qui sert à calculer $\boldsymbol{t}(k+1)$ (on rappelle que $\boldsymbol{J}$ est inversible), le terme $\boldsymbol{K}\boldsymbol{t}(k+1)$ de la deuxième ligne et le terme $\boldsymbol{L}\boldsymbol{t}(k+1)$ de la troisième ligne, alors on retomberait exactement sur la relation (3) pour un State-Space de coefficients $(\boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$. Bien sûr, c'est l'ajout de $\boldsymbol{t}$ qui permet à la SIF, contrairement au State-Space, de décrire n'importe quel ordre pour les calculs d'un algorithme.

La condition *Jprop* peut sembler étrange, mais elle assure que pour un vecteur $\boldsymbol{w}$ donné, on peut calculer $\boldsymbol{v}$ tel que $\boldsymbol{J}\boldsymbol{v} = \boldsymbol{w}$ à l'aide d'un algorithme simple. Par exemple, considérons des vecteurs de taille 3 : $\begin{pmatrix} 1 & 0 & 0 \\ a & 1 & 0 \\ b & c & 1 \end{pmatrix} \begin{pmatrix} v_1 \\ v_2 \\ v_3 \end{pmatrix} = \begin{pmatrix} w_1 \\ w_2 \\ w_3 \end{pmatrix}$. En développant le produit matriciel et en isolant v_i sur la i-ème ligne, on obtient :

$$\begin{aligned} v_1 &= w_1, \\ v_2 &= w_2 - a v_1, \\ v_3 &= w_3 - b v_1 - c v_2. \end{aligned}$$

Le principe de la première ligne de (4) est donc que les composantes de $\boldsymbol{t}(k+1)$ doivent être calculées **dans l'ordre**, chacune pouvant dépendre des précédentes et de $\boldsymbol{x}(k)$ et $\boldsymbol{u}(k)$. Puis, comme pour un State-Space, on calcule le vecteur d'état de l'étape suivante $\boldsymbol{x}(t+1)$ et la sortie $\boldsymbol{y}(k)$, à partir de $\boldsymbol{x}(k)$ et $\boldsymbol{u}(k)$, mais aussi des valeurs intermédiaires de $\boldsymbol{t}(t+1)$ qui viennent d'être obtenues.

Les signaux $\boldsymbol{t}$ et $\boldsymbol{x}$ ont des rôles très différents, et ce n'est pas seulement à cause de la présence de $\boldsymbol{J}$ ni ce qu'elle impose sur le calcul de $\boldsymbol{t}$. Le vecteur d'état $\boldsymbol{x}$ sert à sauvegarder des valeurs d'une étape à la suivante : on calcule sa valeur suivante $\boldsymbol{x}(k+1)$, mais on n'utilise que sa valeur courante $\boldsymbol{x}(k)$ dans les membres droits des égalités. Au contraire, le vecteur intermédiaire $\boldsymbol{t}(t+1)$ est calculé et immédiatement utilisé au cours de la même étape, mais oublié entre chaque étape : on remarque que $\boldsymbol{t}(k)$ n'apparaît pas du tout. L'indice $k+1$ provient de l'écriture implicite de la relation (4) :

$$\begin{pmatrix} \boldsymbol{J} & \boldsymbol{0} & \boldsymbol{0} \\ -\boldsymbol{K} & \boldsymbol{I} & \boldsymbol{0} \\ -\boldsymbol{L} & \boldsymbol{0} & \boldsymbol{I} \end{pmatrix} \begin{pmatrix} \boldsymbol{t}(k+1) \\ \boldsymbol{x}(k+1) \\ \boldsymbol{y}(k) \end{pmatrix} = \begin{pmatrix} \boldsymbol{0} & \boldsymbol{M} & \boldsymbol{N} \\ \boldsymbol{0} & \boldsymbol{P} & \boldsymbol{Q} \\ \boldsymbol{0} & \boldsymbol{R} & \boldsymbol{S} \end{pmatrix} \begin{pmatrix} \boldsymbol{t}(k) \\ \boldsymbol{x}(k) \\ \boldsymbol{u}(k) \end{pmatrix} \tag{5}$$

Sous cette forme, on voit que $\boldsymbol{t}(k)$ ne peut pas être réutilisé à cause des blocs de matrices nulles dans la matrice du membre droit.

Pour formaliser la SIF en Coq, nous utilisons les matrices définies dans [1] à partir de celles de la bibliothèque Coquelicot. `mtx h w` désigne une matrice de dimensions `h`×`w`, dont les composantes ont un type qui dépend du contexte. Ici, ce type sera les réels munis d'une structure d'anneau, ce qu'on explicitera en écrivant `@mtx R_Ring h w`.

Nous définissons d'abord la propriété *Jprop* :

```
Definition J_prop (A : mtx n n) := (forall i, (1 <= i <= n)%Z -> A i i = 1)
                                /\ (forall i j, (1 <= i < j /\ j <= n)%Z -> A i j = 0).
```

Puis, une SIF est un record rassemblant les tailles des signaux vectoriels $\boldsymbol{x}$ et $\boldsymbol{t}$, les neuf matrices dont les tailles se déduisent de l'écriture de (4) (`N_in` étant la taille de l'entrée $\boldsymbol{u}$ et `N_out` celle de la sortie $\boldsymbol{y}$), et la preuve que $\boldsymbol{J}$ vérifie *Jprop* :

```
Record SIF := { SIF_N_x : Z ; (** taille de x **)
                SIF_N_t : Z ; (** taille de t **)
                SIF_J : @mtx R_Ring SIF_N_t SIF_N_t ;
                SIF_K : @mtx R_Ring SIF_N_x SIF_N_t ;
                SIF_L : @mtx R_Ring N_out SIF_N_t ;
                SIF_M : @mtx R_Ring SIF_N_t SIF_N_x ;
                SIF_N : @mtx R_Ring SIF_N_t N_in ;
                SIF_P : @mtx R_Ring SIF_N_x SIF_N_x ;
                SIF_Q : @mtx R_Ring SIF_N_x N_in ;
                SIF_R : @mtx R_Ring N_out SIF_N_x ;
                SIF_S : @mtx R_Ring N_out N_in ;
                SIF_J_prop : J_prop SIF_J           }.
```

Afin de définir le filtre correspondant à une SIF donnée, nous écrivons d'abord une fonction `mtx_inv_J_prop : mtx ?n ?n -> mtx ?n ?n` qui calcule l'inverse d'une matrice qui vérifie *Jprop*. Puis, nous définissons
`SIF_u2x : forall sif : SIF, vect_signal -> vect_signal`, qui à une SIF $(\boldsymbol{J}, \boldsymbol{K}, ..., \boldsymbol{S})$ et un signal $\boldsymbol{u}$ associe le signal $\boldsymbol{x}$ défini par la relation de récurrence :
$\forall k,\ \boldsymbol{x}(k+1) = \boldsymbol{K}(\boldsymbol{J}^{-1}(\boldsymbol{M}\boldsymbol{x}(k) + \boldsymbol{N}\boldsymbol{u}(k))) + \boldsymbol{P}\boldsymbol{x}(k) + \boldsymbol{Q}\boldsymbol{u}(k)$. Enfin, nous construisons $\boldsymbol{y}$ comme dans (4). Nous associons ainsi à une SIF une fonction qui transforme un signal d'entrée $\boldsymbol{u}$ en un signal de sortie $\boldsymbol{y}$, c'est-à-dire un filtre : `MIMO_filter_from_SIF : SIF -> MIMO_filter`.

3.2 Exemple : détailler l'ordre des calculs à l'aide d'une SIF

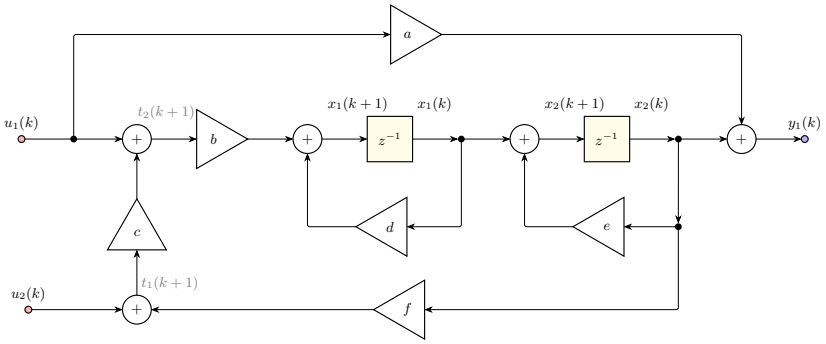


Fig. 3: Graphe à flot de données de l'exemple.

Considérons le filtre $\mathscr{H}$ défini par le graphe à flot de données de la figure 3. Il prend une entrée $\boldsymbol{u} = \begin{pmatrix} u_1 \\ u_2 \end{pmatrix}$ de taille 2 et produit une sortie $\boldsymbol{y} = \begin{pmatrix} y_1 \end{pmatrix}$ de taille 1

selon les équations suivantes, où x_1 et x_2 sont des signaux auxiliaires (tous les signaux étant nuls pour $k < 0$) :

$$\begin{aligned} x_1(k+1) &= b(u_1(k) + c(u_2(k) + fx_2(k))) + dx_1(k), \\ x_2(k+1) &= x_1(k) + ex_2(k), \\ y(k) &= au_1(k) + x_2(k). \end{aligned} \tag{6}$$

Dans un premier temps, traduisons ce filtre vers un State-Space $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$. Cela permettra à la fois de s'habituer au raisonnement avec une construction plus simple que celle d'une SIF, et d'exposer les limites du State-Space. Les $x_1(k)$ et $x_2(k)$ sont exactement les informations que nous avons besoin de retenir d'une étape sur l'autre, donc le vecteur d'état sera $\boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$. D'après (3), nous devons avoir :

$$\begin{pmatrix} bu_1(k) + bcu_2(k) + bcfx_2(k) + dx_1(k) \\ x_1(k) + ex_2(k) \end{pmatrix} = \boldsymbol{x}(k+1) = \boldsymbol{A} \begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix} + \boldsymbol{B} \begin{pmatrix} u_1(k) \\ u_2(k) \end{pmatrix},$$

$$\big(au_1(k) + x_2(k)\big) = \boldsymbol{y}(k) = \boldsymbol{C} \begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix} + \boldsymbol{D} \begin{pmatrix} u_1(k) \\ u_2(k) \end{pmatrix},$$

donc $\boldsymbol{A} = \begin{pmatrix} d & bcf \\ 1 & e \end{pmatrix}$, $\boldsymbol{B} = \begin{pmatrix} b & bc \\ 0 & 0 \end{pmatrix}$, $\boldsymbol{C} = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ et $\boldsymbol{D} = \begin{pmatrix} a \\ 0 \end{pmatrix}$.

Le problème, c'est que nous avons supposé que $b(u_1(k)+c(u_2(k)+fx_2(k)))+dx_1(k) = bu_1(k) + bcu_2(k) + bcfx_2(k) + dx_1(k)$, ce qui n'est pas toujours vrai en précision finie. Pour vraiment transcrire l'algorithme associé au graphe de la figure 3, nous avons besoin de prendre en compte l'ordre des opérations, et c'est exactement ce que fait la SIF.

Traduisons maintenant ce filtre vers une SIF. Pour pouvoir décrire correctement le calcul de $x_1(k+1)$, nous introduisons un signal de valeurs intermédiaires $\boldsymbol{t}$ de taille 2, tel que :

$$\begin{aligned} &t_1(k+1) = u_2(k) + fx_2(k), \\ &t_2(k+1) = u_1(k) + ct_1(k+1) \quad i.e. \quad -ct_1(k+1) + t_2(k+1) = u_1(k), \\ &x_1(k+1) = bt_2(k+1) + dx_1(k). \end{aligned}$$

Nous devons alors avoir, en appliquant (4) :

$$\boldsymbol{J} \begin{pmatrix} u_2(k) + fx_2(k) \\ u_1(k) + ct_1(k+1) \end{pmatrix} = \boldsymbol{M} \begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix} + \boldsymbol{N} \begin{pmatrix} u_1(k) \\ u_2(k) \end{pmatrix},$$

$$\begin{pmatrix} bt_2(k+1) + dx_1(k) \\ x_1(k) + ex_2(k) \end{pmatrix} = \boldsymbol{K} \begin{pmatrix} t_1(k+1) \\ t_2(k+1) \end{pmatrix} + \boldsymbol{P} \begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix} + \boldsymbol{Q} \begin{pmatrix} u_1(k) \\ u_2(k) \end{pmatrix},$$

$$\big(au_1(k) + x_2(k)\big) = \boldsymbol{L} \begin{pmatrix} t_1(k+1) \\ t_2(k+1) \end{pmatrix} + \boldsymbol{R} \begin{pmatrix} x_1(k) \\ x_2(k) \end{pmatrix} + \boldsymbol{S} \begin{pmatrix} u_1(k) \\ u_2(k) \end{pmatrix}.$$

Nous obtenons $\boldsymbol{J} = \begin{pmatrix} 1 & 0 \\ -c & 1 \end{pmatrix}$, $\boldsymbol{M} = \begin{pmatrix} 0 & f \\ 0 & 0 \end{pmatrix}$, $\boldsymbol{N} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$, $\boldsymbol{K} = \begin{pmatrix} 0 & b \\ 0 & 0 \end{pmatrix}$, $\boldsymbol{P} = \begin{pmatrix} d & 0 \\ 1 & e \end{pmatrix}$, $\boldsymbol{Q} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$, $\boldsymbol{L} = \begin{pmatrix} 0 & 0 \end{pmatrix}$, $\boldsymbol{R} = \begin{pmatrix} 0 & 1 \end{pmatrix}$ et $\boldsymbol{S} = \begin{pmatrix} a & 0 \end{pmatrix}$. La SIF ainsi construite

décrit exactement le graphe de la figure 3, y compris l'ordre des opérations. La traduction de n'importe quel graphe à flot de données vers une SIF est présentée dans [11]. Elle n'est cependant pas formalisée en Coq, car les graphes eux-mêmes seraient longs et complexes à définir.

3.3 Traduction d'un State-Space vers une SIF

Nous avons vu que la SIF peut représenter n'importe quel algorithme. Cependant, pour pouvoir analyser formellement l'impact des erreurs d'arrondi sur un algorithme donné, il nous faut une traduction certifiée de cet algorithme vers une SIF. Nous avons formalisé une telle traduction pour le State-Space présenté dans la section 2.3, qui est souvent utilisé par la communauté de l'automatique.

Soit $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$ un State-Space, que nous souhaitons transformer en SIF. Nous avons vu en section 3.1 que si on enlève toutes les apparitions de $\boldsymbol{t}$ dans les équations (4), c'est-à-dire la première équation entière et un terme de chacune des deux autres, on retombe sur les équations (3) correspondant à un State-Space $(\boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$. Il suffit donc de définir :

$$\begin{aligned} \boldsymbol{K} &= \boldsymbol{0}, & \boldsymbol{L} &= \boldsymbol{0}, \\ \boldsymbol{P} &= \boldsymbol{A}, & \boldsymbol{Q} &= \boldsymbol{B}, \\ \boldsymbol{R} &= \boldsymbol{C}, & \boldsymbol{S} &= \boldsymbol{D}. \end{aligned} \tag{7}$$

Comme les termes en $\boldsymbol{t}$ ont été annulés en même temps que les matrices $\boldsymbol{K}$ et $\boldsymbol{L}$, on peut définir n'importe quelles valeurs pour $\boldsymbol{J}$, $\boldsymbol{M}$ et $\boldsymbol{N}$, par exemple l'identité pour $\boldsymbol{J}$ (qui doit tout de même vérifier la propriété *Jprop*) et encore des matrices nulles pour les deux autres. La SIF $(\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$ ainsi obtenue décrit le même filtre que le State-Space $(\boldsymbol{A}, \boldsymbol{B}, \boldsymbol{C}, \boldsymbol{D})$.

Cette méthode fonctionne en choisissant n'importe quelle taille pour $\boldsymbol{t}$. On peut faire plus simple et fixer la taille de $\boldsymbol{t}$ à zéro, ce qui évite d'avoir à préciser des valeurs pour $\boldsymbol{J}$, $\boldsymbol{K}$, $\boldsymbol{L}$, $\boldsymbol{M}$ et $\boldsymbol{N}$, qui ont chacune au moins une dimension nulle. Néanmoins, dans notre formalisation en Coq, nous avons choisi de donner la taille 1 à $\boldsymbol{t}$, au cas où nous aurions besoin plus tard de propriétés qui se comportent différemment sur des matrices ayant une dimension nulle.

En Coq, nous définissons la transformation ci-dessus, et nous prouvons que la SIF obtenue décrit bien le même filtre que le State-Space de départ :

```
Definition SIF_from_StateSpace : StateSpace -> SIF := ...
Theorem SIF_from_StateSpace_same_filter : forall stsp : StateSpace,
 MIMO_filter_from_SIF (SIF_from_StateSpace stsp) = MIMO_filter_from_StSp stsp.
```

En combinant la transformation ci-dessus et celles vers un State-Space formalisées dans [1], nous pouvons décrire explicitement sous forme de SIF deux autres réalisations usuelles présentées dans la section 2 : la forme directe I et la forme directe II. Cependant, l'ordre des opérations est celui imposé par le State-Space intermédiaire, et n'est pas forcément identique à celui de l'algorithme d'origine. Pour étudier les erreurs d'arrondi dans ces algorithmes exacts, il faudra les traduire indépendamment en SIF.

3.4 Traduction d'une SIF vers un State-Space

Intéressons-nous maintenant à la transformation inverse de celle vue précédemment : d'une SIF à un State-Space. Elle permet d'obtenir une représentation plus simple avec seulement quatre matrices au lieu de neuf. L'inconvénient est bien sûr que comme le State-Space est moins expressif, l'ordre des calculs risque d'être modifié. Cependant, ceci n'est pas un problème en précision infinie. Cette transformation reste donc intéressante pour étudier le filtre modèle exprimé avec des nombres réels.

Soit $(\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$ une SIF. Le filtre correspondant peut être également décrit par le State-Space $(\boldsymbol{A_Z}, \boldsymbol{B_Z}, \boldsymbol{C_Z}, \boldsymbol{D_Z})$ où :

$$\begin{aligned} \boldsymbol{A_Z} &= \boldsymbol{K}\boldsymbol{J}^{-1}\boldsymbol{M} + \boldsymbol{P}, & \boldsymbol{B_Z} &= \boldsymbol{K}\boldsymbol{J}^{-1}\boldsymbol{N} + \boldsymbol{Q}, \\ \boldsymbol{C_Z} &= \boldsymbol{L}\boldsymbol{J}^{-1}\boldsymbol{M} + \boldsymbol{R}, & \boldsymbol{D_Z} &= \boldsymbol{L}\boldsymbol{J}^{-1}\boldsymbol{N} + \boldsymbol{S}. \end{aligned} \tag{8}$$

On rappelle que $\boldsymbol{J}$ vérifie la propriété *Jprop*, ce qui implique qu'elle est inversible et que son inverse est facile à calculer. L'indice Z est couramment utilisé pour marquer le lien avec une SIF.

Nous définissons cette transformation en Coq, et prouvons que le State-Space construit décrit le même filtre que la SIF d'origine :

```
Definition StateSpace_from_SIF : SIF -> StateSpace := ...
Theorem StateSpace_SIF_same_filter : forall sif : SIF,
 MIMO_filter_from_StSp (StateSpace_from_SIF sif) = MIMO_filter_from_SIF sif.
```

4 Filtre d'erreur appliqué à la SIF

Nous nous intéressons ici à la propagation des erreurs d'arrondi, c'est-à-dire l'effet des erreurs locales dans chaque calcul sur les sorties du filtre. Cela s'applique à n'importe quelle arithmétique en précision finie. Cette propagation est décrite par le théorème du filtre d'erreur. Ce théorème est déjà présenté et prouvé dans [1] pour un filtre défini par un State-Space. Nous le prouvons ici dans le cas d'une SIF, qui est plus expressive.

Soit $(\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$ une SIF. On appelle $\mathscr{H}$ le filtre correspondant, calculé avec une précision infinie. Les signaux $\boldsymbol{t}$, $\boldsymbol{x}$ et $\boldsymbol{y}$ sont respectivement le vecteur intermédiaire, le vecteur d'état et la vecteur de sortie, obtenus en (4), toujours avec une précision infinie. Soit $\boldsymbol{t}^*$, $\boldsymbol{x}^*$ et $\boldsymbol{y}^*$ ces mêmes éléments obtenus en appliquant (4) en précision finie, et $\mathscr{H}^*$ le filtre implémenté, qui à l'entrée $\boldsymbol{u}$ associe la sortie $\boldsymbol{y}^*$. L'erreur finale qui nous intéresse est alors $\boldsymbol{\Delta y}(k) = \boldsymbol{y}^*(k) - \boldsymbol{y}(k)$.

Dans le filtre implémenté $\mathscr{H}_\varepsilon$, les relations (4) ne sont respectées qu'aux erreurs d'arrondi près. On représente cela par l'ajout de termes d'erreurs $\boldsymbol{\varepsilon_t}(k)$, $\boldsymbol{\varepsilon_x}(k)$ et $\boldsymbol{\varepsilon_y}(k)$:

$$\begin{aligned} \boldsymbol{J}\boldsymbol{t}^*(k+1) &= \boldsymbol{M}\boldsymbol{x}^*(k) + \boldsymbol{N}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_t}(k) \\ \boldsymbol{x}^*(k+1) &= \boldsymbol{K}\boldsymbol{t}^*(k+1) + \boldsymbol{P}\boldsymbol{x}^*(k) + \boldsymbol{Q}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_x}(k) \\ \boldsymbol{y}^*(k) &= \boldsymbol{L}\boldsymbol{t}^*(k+1) + \boldsymbol{R}\boldsymbol{x}^*(k) + \boldsymbol{S}\boldsymbol{u}(k) + \boldsymbol{\varepsilon_y}(k) \end{aligned} \tag{9}$$

On note $\boldsymbol{\varepsilon}(k)$ la concaténation de ces vecteurs d'erreur : $\boldsymbol{\varepsilon}(k) = \begin{pmatrix} \boldsymbol{\varepsilon_t}(k) \\ \boldsymbol{\varepsilon_x}(k) \\ \boldsymbol{\varepsilon_y}(k) \end{pmatrix}$.
On a ainsi défini un signal vectoriel $\boldsymbol{\varepsilon}$ contenant toutes les erreurs locales.

Théorème (Filtre d'erreur). *Soit $(\boldsymbol{J}, \boldsymbol{K}, \boldsymbol{L}, \boldsymbol{M}, \boldsymbol{N}, \boldsymbol{P}, \boldsymbol{Q}, \boldsymbol{R}, \boldsymbol{S})$ une SIF, décrivant un filtre modèle $\mathscr{H}$ en précision infinie. Soit $\mathscr{H}^*$ le filtre obtenu en appliquant l'algorithme correspondant en précision finie. On note $\boldsymbol{y}$ la sortie théorique de $\mathscr{H}$, $\boldsymbol{y}^*$ la sortie calculée de $\mathscr{H}^*$, et $\boldsymbol{\Delta y} = \boldsymbol{y}^* - \boldsymbol{y}$ l'erreur finale de l'implémentation par rapport au modèle. Enfin, soit $\boldsymbol{\varepsilon}$ le signal vectoriel des erreurs locales défini ci-dessus. Alors, le signal $\boldsymbol{\Delta y}$ s'obtient comme la sortie correspondant à l'entrée $\boldsymbol{\varepsilon}$ à travers le filtre d'erreur $\mathscr{H}_\varepsilon$, défini comme le filtre de State-Space $(\boldsymbol{A_Z}, \boldsymbol{B}_\varepsilon, \boldsymbol{C_Z}, \boldsymbol{D}_\varepsilon)$, où :*

$$\boldsymbol{B}_\varepsilon = (\boldsymbol{K}\ \boldsymbol{J}^{-1}\ \boldsymbol{I}_n\ \boldsymbol{0}), \qquad \boldsymbol{D}_\varepsilon = (\boldsymbol{L}\ \boldsymbol{J}^{-1}\ \boldsymbol{0}\ \boldsymbol{I}_p), \tag{10}$$

(on rappelle que $\boldsymbol{J}^{-1}$ est facile à calculer grâce à Jprop), et $\boldsymbol{A_Z}$ et $\boldsymbol{C_Z}$ sont les matrices du State-Space associé à la SIF $(\boldsymbol{J}, \boldsymbol{K}, ..., \boldsymbol{S})$, définies par l'équation (8) *:*

$$\boldsymbol{A_Z} = \boldsymbol{K}\boldsymbol{J}^{-1}\boldsymbol{M} + \boldsymbol{P}, \qquad \boldsymbol{C_Z} = \boldsymbol{L}\boldsymbol{J}^{-1}\boldsymbol{M} + \boldsymbol{R}. \qquad \text{(rappel de (8))}$$

De manière équivalente, comme illustré par la figure 4, la sortie $\boldsymbol{y}^*$ calculée en pratique est égale à la somme de la sortie $\boldsymbol{y}$ du filtre idéal $\mathscr{H}$ en précision infinie, et de la sortie $\boldsymbol{\Delta y}(k)$ du filtre d'erreur $\mathscr{H}_\varepsilon$ auquel on donne en entrée le signal d'erreurs locales $\boldsymbol{\varepsilon}$.

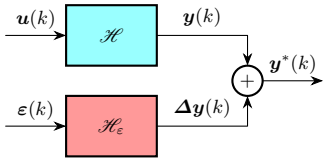


Fig. 4: Décomposition du filtre implémenté en modèle $\mathscr{H}$ et filtre d'erreur $\mathscr{H}_\varepsilon$

En Coq, nous considérons une SIF `sif`, un signal vectoriel d'entrée `u`, ainsi que des signaux vectoriels d'erreurs `err_t`, `err_x` et `err_y`, de tailles correspondant à celles de `sif`. Ces éléments sont bien sûr quantifiés universellement. Nous définissons récursivement le signal `x'`, puis le signal `y'`, qui correspondent respectivement à $\boldsymbol{x}^*$ et $\boldsymbol{y}^*$ obtenus en appliquant (9). Nous calculons également `B_err` et `D_err` données par (10). Nous pouvons alors définir `stsp_err` comme le State-Space dont les matrices $\boldsymbol{B}$ et $\boldsymbol{D}$ sont `B_err` et `D_err`, et dont les matrices $\boldsymbol{A}$ et $\boldsymbol{C}$ sont celles du State-Space `stsp_model` := `StateSpace_from_SIF sif`, obtenu en appliquant la transformation de la section 3.4 à `sif`. Enfin, nous prouvons que la sortie `y'` calculée en pratique est égale à la somme de la sortie du filtre théorique `h_model` décrit par `sif`, et de la sortie du filtre d'erreur `h_err` décrit par `stsp_err`

pour l'entrée `err` qui est la concaténation des vecteurs d'erreurs locales `err_t`, `err_x` et `err_y` :

```
Definition h_model : MIMO_filter := MIMO_filter_from_SIF sif.
Definition h_err : MIMO_filter := MIMO_filter_from_StSp stsp_err.
Definition err := vect_signal_concat err_t (vect_signal_concat err_x err_y).
Theorem actual_is_model_plus_err :
    y' = vect_signal_plus (h_model u) (h_err err).
```

Ceci est exactement la formulation équivalente décrite par la figure 4 du théorème du filtre d'erreur.

La caractérisation de l'erreur finale $\boldsymbol{\Delta y}$ fournie par le théorème du filtre d'erreur va ensuite permettre de borner cette erreur. En effet, il existe un théorème qui permet de borner la sortie d'un filtre à partir d'une borne sur son entrée. Ce théorème, appelé le *Worst-Case Peak-Gain*, est prouvé en Coq dans [1]. Comme $\boldsymbol{\Delta y}$ est la sortie à travers le filtre d'erreur $\mathscr{H}_\varepsilon$ correspondant à l'entrée $\boldsymbol{\varepsilon}$, il suffit de borner le vecteur d'erreurs locales $\boldsymbol{\varepsilon}$. Or, les arithmétiques en précision finie usuelles garantissent l'arrondi correct des opérations, c'est-à-dire que le résultat d'une seule opération est la valeur représentable la plus proche de la valeur théorique. De plus, ces arithmétiques fournissent généralement une opération "somme de produits" qui permet de calculer une valeur de la forme $\Sigma_{1\leq i\leq N}\ a_i b_i$ sans arrondis intermédiaires, ce qui est très utile pour les produits vectoriels et matriciels. Les erreurs locales contenues dans $\boldsymbol{\varepsilon}$ proviennent donc chacune d'un très petit nombre d'opérations, et peuvent être bornées explicitement pour une arithmétique donnée. On connaît alors une borne sur le signal d'entrée fourni à $\mathscr{H}_\varepsilon$, de laquelle on déduit une borne sur sa sortie $\boldsymbol{\Delta y}$.

5 Conclusion

Nous avons formalisé en Coq la SIF, qui permet d'exprimer tous les algorithmes de filtres numériques en précisant dans quel ordre sont effectuées les opérations. Ce dernier point est particulièrement pertinent en précision finie : modifier l'ordre des calculs entraine des erreurs d'arrondi différentes, ce qui peut donner des résultats de qualité très variable. Nous avons défini en Coq la traduction du State-Space vers la SIF, et prouvé que le filtre qu'elles décrivent est conservé par ces traductions. Cela nous permet d'exprimer sous forme de SIF d'autre réalisations usuelles, les formes directes I et II, qui ont déjà été traduites en Coq vers un State-Space [1]. Enfin, nous avons prouvé formellement le théorème du filtre d'erreur dans le contexte de la SIF, établissant ainsi une caractérisation certifiée de la propagation des erreurs d'arrondi pour n'importe quel algorithme traduit en Coq vers une SIF.

Les preuves Coq ne présentent pas de difficulté particulière une fois que nous disposons des bons lemmes pour raisonner par récurrence forte sur $\mathbb{Z}$ (généralement initialisée pour tout $k < 0$) et surtout pour développer les calculs matriciels. La plupart de ces lemmes ont déjà été établis dans [1]. Notamment, nous utilisons les matrices introduites dans [1], basées sur celles de Coquelicot [6] mais dont

les indices sont des entiers relatifs. Une valeur par défaut est renvoyée pour des indices excédant les bornes, ce qui inclut tous les entiers négatifs. Ces matrices sont utilisées à travers une interface qui est indépendante de leur construction. L'intérêt est que cela nous permet de changer facilement la bibliothèque sur laquelle nous nous appuyons. Notamment, nous envisageons d'utiliser les matrices de la bibliothèque Mathcomp [12], afin de s'appuyer sur ses théorèmes d'algèbre linéaire pour calculer précisément la borne donnée par le théorème du *Worst-Case Peak-Gain* mentionné à la fin de la section 4.

Une perspective naturelle est la prise en compte du choix d'arithmétique en précision finie. En effet, le théorème du filtre d'erreur s'applique à n'importe laquelle d'entre elles : il exprime l'erreur finale en fonction des erreurs locales à chaque étape, indépendamment des caractéristiques de ces dernières. Tenir compte des restrictions imposées par un standard donné sur les erreurs locales permettra donc d'obtenir plus d'informations sur l'erreur finale. Le cas de l'arithmétique en virgule flottante pourra s'appuyer sur la bibliothèque Flocq [13]. Cependant, les filtres numériques présents dans les systèmes embarqués utilisent surtout l'arithmétique en virgule fixe. Celle-ci est également formalisée dans Flocq, mais il manque à cette bibliothèque deux éléments pour pouvoir effectuer une étude approfondie des filtres en virgule fixe. D'une part, ces filtres utilisent généralement la représentation du complément à deux, tandis que Flocq a choisi une autre convention usuelle qui représente les nombres négatifs différemment. D'autre part, il existe différents comportements en cas d'*overflow* : par exemple, utiliser un modulo pour toujours avoir une valeur dans le bon intervalle, ou encore la règle de saturation, qui ramène un résultat trop grand à la valeur maximale autorisée. Dans certains cas, nous souhaiterons étudier formellement chacun de ces comportements utilisés en pratique. Dans d'autres cas, nous voudrons prouver que dans un algorithme donné, aucun *overflow* ne peut arriver. Ainsi, la première étape de l'étude approfondie des filtres des systèmes embarqués consistera à ajouter à Flocq le complément à deux et diverses options de gestion des *overflows*.

Dans un contexte industriel, comme les systèmes de contrôle en automobile ou en aéronautique, un filtre cible est spécifié, par exemple sous la forme d'une fonction de transfert souhaitée. Puis une réalisation est sélectionnée, c'est-à-dire un algorithme, ainsi qu'un support matériel, sur lequel il sera implémenté selon une arithmétique en précision finie bien précise. Nous décrivons alors l'algorithme sous forme de SIF et appliquons le théorème du filtre d'erreur, puis d'autres étapes qu'il reste encore à formaliser, notamment pour tenir compte de l'arithmétique choisie. L'objectif est d'obtenir une borne certifiée sur l'erreur finale causée par la précision finie sur la sortie du filtre. Une perspective complémentaire sera la génération automatique de code certifié correspondant à une SIF, donc à n'importe quel algorithme traduit formellement vers la SIF. L'étape suivante consistera à automatiser la sélection de l'algorithme, dans le but de minimiser l'erreur finale résultant des erreurs d'arrondi. On pourra s'appuyer sur des techniques de transformation automatique permettant d'améliorer la précision numérique d'un programme [14]. Cependant, ces techniques généralistes ne

pourront peut-être pas explorer d'autres transformations du type changement de base d'une SIF [4], qui ont également pour but d'améliorer le comportement numérique en précision finie. La complémentarité des deux méthodes sera une piste d'étude intéressante.

Références

1. Gallois-Wong, D., Boldo, S., Hilaire, T.: A coq formalization of digital filters. In Rabe, F., Farmer, W.M., Passmore, G.O., Youssef, A., eds.: Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings. Volume 11006 of Lecture Notes in Computer Science., Springer (2018) 87–103
2. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development. Texts in Theoretical Computer Science. Springer (2004)
3. The Coq Development Team: The Coq Proof Assistant Reference Manual. (2017)
4. Hilaire, T., Chevrel, P., Whidborne, J.: A unifying framework for finite wordlength realizations. IEEE Trans. on Circuits and Systems **8**(54) (August 2007) 1765–1774
5. Mayero, M.: Formalisation et automatisation de preuves en analyses réelle et numérique. PhD thesis, Université Paris VI (décembre 2001)
6. Boldo, S., Lelay, C., Melquiond, G.: Coquelicot: A user-friendly library of real analysis for Coq. Mathematics in Computer Science **9**(1) (2015) 41–62
7. Akbarpour, B., Tahar, S.: Error analysis of digital filters using hol theorem proving. Journal of Applied Logic **5**(4) (2007) 651–666 from the 4th International Workshop on Computational Models of Scientific Reasoning and Applications.
8. Akbarpour, B., Tahar, S., Dekdouk, A.: Formalization of fixed-point arithmetic in HOL. Formal Methods in System Design **27**(1) (Sep 2005) 173–200
9. Park, J., Pajic, M., Sokolsky, O., Lee, I. In: Automatic Verification of Finite Precision Implementations of Linear Controllers. Springer Berlin Heidelberg, Berlin, Heidelberg (2017) 153–169
10. Lopez, B.: Implémentation optimale de filtres linéaires en arithmétique virgule fixe. PhD thesis, Université Pierre et Marie Curie, Sorbonne Université (Nov. 2014)
11. Hilaire, T., Volkova, A., Ravoson, M.: Reliable fixed-point implementation of linear data-flows. In: Proc. IEEE Workshop on Signal Processing Systems (SiPS). (2016)
12. Gonthier, G., Asperti, A., Avigad, J., Bertot, Y., Cohen, C., Garillot, F., Le Roux, S., Mahboubi, A., O'Connor, R., Ould Biha, S., Pasca, I., Rideau, L., Solovyev, A., Tassi, E., Théry, L.: A Machine-Checked Proof of the Odd Order Theorem. In Blazy, S., Paulin, C., Pichardie, D., eds.: 4th Conference on Interactive Theorem Proving. Volume 7998 of LNCS., France, Springer (2013) 163–179
13. Boldo, S., Melquiond, G.: Computer Arithmetic and Formal Proofs. ISTE Press - Elsevier (December 2017)
14. Damouche, N., Martel, M., Chapoutot, A.: Improving the numerical accuracy of programs by automatic transformation. International Journal on Software Tools for Technology Transfer (September 2016)